(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 152 329 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
07.11.2001 Bulletin 2001/45

(51) Int Cl.7: G06F 9/38

(21) Application number: 01302951.7

(22) Date of filing: 29.03.2001

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE TR
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 30.03.2000 US 538757

(71) Applicant: Agere Systems Guardian Corporation
Orlando, Florida 32819 (US)

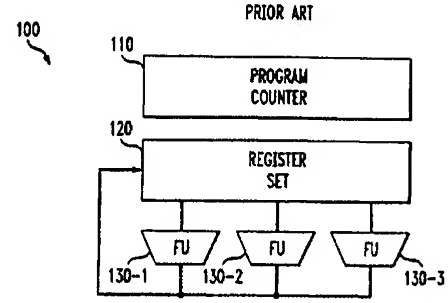
(72) Inventors:
- Berenbaum, Alan David
New York, NY 10011 (US)
- Heintze, Nevin
Morristown, NJ 07960 (US)
- Jeremiassen, Tor E.
Somerset, NJ 08873 (US)
- Kaxiras, Stefanos
Jersey City, NJ 07302 (US)

(74) Representative: Williams, David John et al
Page White & Farrer,
54 Doughty Street
London WC1N 2LS (GB)

(54) Method and apparatus for identifying splittable packets in a multithreaded vliw processor

(57) A method and apparatus are disclosed for allocating functional units in a multithreaded very large instruction word (VLIW) processor. The present invention combines the techniques of conventional very long instruction word (VLIW) architectures and conventional multithreaded architectures to reduce execution time within an individual program, as well as across a workload. The present invention utilizes instruction packet splitting to recover some efficiency lost with conventional multithreaded architectures. Instruction packet splitting allows an instruction bundle to be partially issued in one cycle, with the remainder of the bundle issued during a subsequent cycle. There are times, however, when instruction packets cannot be split without violating the semantics of the instruction packet assembled by the compiler. A packet split identification bit is disclosed that allows hardware to efficiently determine when it is permissible to split an instruction packet. The split bit informs the hardware when splitting is prohibited. The allocation hardware assigns as many instructions from each packet as will fit on the available functional units, rather than allocating all instructions in an instruction packet at one time, provided the split bit has not been set. Those instructions that cannot be allocated to a functional units are retained in a ready-to-run register. On subsequent cycles, instruction packets in which all instructions have been issued to functional units are updated from their thread's instruction stream, while instruction packets with instructions that have been held are retained. The functional unit allocation logic can then assign instructions from the newly-loaded instruction packets as well as instructions that were not issued from the retained instruction packets.

FIG. 1
PRIOR ART



EP 1 152 329 A1

Description

Cross-Reference to Related Applications

[0001] The present invention is related to United States Patent Application entitled "Method and Apparatus for Allocating Functional Units in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 7-2-3-3); United States Patent Application entitled "Method and Apparatus for Releasing Functional Units in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 8-3-4-4); and United States Patent Application entitled "Method and Apparatus for Splitting Packets in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 9-4-5-5), each filed contemporaneously herewith, assigned to the assignee of the present invention and incorporated by reference herein.

Field of the Invention

[0002] The present invention relates generally to multithreaded processors, and, more particularly, to a method and apparatus for splitting packets in such multithreaded processors.

Background of the Invention

[0003] Computer architecture designs attempt to complete workloads more quickly. A number of architecture designs have been proposed or suggested for exploiting program parallelism. Generally, an architecture that can issue more than one operation at a time is capable of executing a program faster than an architecture that can only issue one operation at a time. Most recent advances in computer architecture have been directed towards methods of issuing more than one operation at a time and thereby speed up the operation of programs. FIG. I illustrates a conventional microprocessor architecture 100. Specifically, the microprocessor 100 includes a program counter (PC) 110, a register set 120 and a number of functional units (FUs) 130-N. The redundant functional units (FUs) 130-1 through 130-N provide the illustrative microprocessor architecture 100 with sufficient hardware resources to perform a corresponding number of operations in parallel.

[0004] An architecture that exploits parallelism in a program issues operands to more than one functional unit at a time to speed up the program execution. A number of architectures have been proposed or suggested with a parallel architecture, including superscalar processors, very long instruction word (VLIW) processors and multithreaded processors, each discussed below in conjunction with FIGS. 2, 4 and 5, respectively. Generally, a superscalar processor utilizes hardware at run-time to dynamically determine if a number of operations from a single instruction stream are independent, and if so, the processor executes the instructions using parallel arithmetic and logic units (ALUs). Two instructions are said to be independent if none of the source operands are dependent on the destination operands of any instruction that precedes them. A very long instruction word (VLIW) processor evaluates the instructions during compilation and groups the operations appropriately, for parallel execution, based on dependency information. A multithreaded processor, on the other hand, executes more than one instruction stream in parallel, rather than attempting to exploit parallelism within a single instruction stream.

[0005] A superscalar processor architecture 200, shown in FIG. 2, has a number of functional units that operate independently, in the event each is provided with valid data. For example, as shown in FIG. 2, the superscalar processor 200 has three functional units embodied as arithmetic and logic units (ALUs) 230-N, each of which can compute a result at the same time. The superscalar processor 200 includes a front-end section 208 having an instruction fetch block 210, an instruction decode block 215, and an instruction sequencing unit 220 (issue block). The instruction fetch block 210 obtains instructions from an input queue 205 of a single threaded instruction stream. The instruction sequencing unit 220 identifies independent instructions that can be executed simultaneously in the available arithmetic and logic units (ALUs) 230-N, in a known manner. The refine block 250 allows the instructions to complete, and also provides buffering and reordering for writing results back to the register set 240.

[0006] In the program fragment 310 shown in FIG. 3, instructions in locations L1, L2 and L3 are independent, in that none of the source operands in instructions L2 and L3 are dependent on the destination operands of any instruction that precedes them. When the program counter (PC) is set to location L1, the instruction sequencing unit 220 will look ahead in the instruction stream and detect that the instructions at L2 and L3 are independent, and thus all three can be issued simultaneously to the three available functional units 230-N. For a more detailed discussion of superscalar processors, see, for example, James. E. Smith and Gurindar. S. Sohi, "The Microarchitecture of Superscalar Processors," Proc. of the IEEE (Dec. 1995), incorporated by reference herein.

[0007] As previously indicated, a very long instruction word (VLIW) processor 400, shown in FIG.4, relies on software to detect data parallelism at compile time from a single instruction stream, rather than using hardware to dynamically detect parallelism at run time. A VLIW compiler, when presented with the source code that was used to generate the code fragment 310 in FIG. 3, would detect the instruction independence and construct a single, very long instruction comprised of all three operations. At run time, the issue logic of the processor 400 would issue this wide instruction in one cycle, directing data to all available functional units 430-N. As shown in

FIG. 4, the very long instruction word (VLIW) processor 400 includes an integrated fetch/decode block 420 that obtains the previously grouped instructions 410 from memory. For a more detailed discussion of very long instruction word (VLIW) processors, see, for example, Burton J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," SPIE Real Time Signal Procesing IV, 241-248 (1981), incorporated by reference herein.

[0008] One variety of VLIW processors, for example, represented by the Multiflow architecture, discussed in Robert P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," IEEE Transactions on Computers (August 1988), uses a fixed-width instruction, in which predefined fields direct data to all functional units 430-N at once. When all operations specified in the wide instruction are completed, the processor issues a new, multi-operation instruction. Some more recent VLIW processors, such as the C6x processor commercially available from Texas Instruments, of Dallas, TX and the EPIC IA-64 processor commercially available from Intel Corp, of Santa Clara, CA, instead use a variable-length instruction packet, which contains one or more operations bundled together.

[0009] A multithreaded processor 500, shown in FIG. 5, gains performance improvements by executing more than one instruction stream in parallel, rather than attempting to exploit parallelism within a single instruction stream. The multithreaded processor 500 shown in FIG. 5 includes a program counter 510-N, a register set 520-N and a functional unit 530-N, each dedicated to a corresponding instruction stream N. Alternate implementations of the multithreaded processor 500 have utilized a single functional unit 530, with several register sets 520-N and program counters 510-N. Such alternate multithreaded processors 500 are designed in such a way that the processor 500 can switch instruction issue from one program counter/register set 510-N/520-N to another program counter/register set 510-N/520-N in one or two cycles. A long latency instruction, such as a LOAD instruction, can thus be overlapped with shorter operations from other instruction streams. The TERA MTA architecture, commercially available from Tera Computer Company, of Seattle, WA, is an example of this type.

[0010] An extension of the multithreaded architecture 500, referred to as Simultaneous Multithreading, combines the superscalar architecture, discussed above in conjunction with FIG. 2, with the multithreaded designs, discussed above in conjunction with FIG. 5. For a detailed discussion of Simultaneous Multithreading techniques, see, for example, Dean Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism, " Proc. of the 22nd Annual Int'l Symposium on Computer Architecture, 392-403 (Santa Margherita Ligure, Italy, June 1995), incorporated by reference herein. Generally, in a Simultaneous Multithreading architecture, there is a pool of functional units, any number of which may be dynamically assigned to an instruction which can issue from any one of a number of program counter/register set structures. By sharing the functional units among a number of program threads, the Simultaneous Multithreading architecture can make more efficient use of hardware than that shown in FIG. 5.

[0011] While the combined approach of the Simultaneous Multithreading architecture provides improved efficiency over the individual approaches of the superscalar architecture or the multithreaded architecture, Simultaneous Multithreaded architectures still require elaborate issue logic to dynamically examine instruction streams in order to detect potential parallelism. In addition, when an operation takes multiple cycles, the instruction issue logic may stall, because there is no other source of operations available. Conventional multithreaded processorss issue instructions from a set of instructions simultaneously, with the functional units designed to accommodate the widest potential issue. A need therefore exists for a multithreaded processor architecture that does not require a dynamic determination of whether or not two instruction streams are independent. A further need exists for a multithreaded architecture that provides simultaneous multithreading. Yet another need exists for a method and apparatus that improve the utilization of processor resources for each cycle.

Summary of the Invention

[0012] Generally, a method and apparatus are disclosed for allocating functional units in a multithreaded very large instruction word (VLIW) processor. The present invention combines the techniques of conventional very long instruction word (VLIW) architectures and conventional multithreaded architectures. The combined architecture of the present invention reduces execution time within an individual program, as well as across a workload. The present invention utilizes instruction packet splitting to recover some efficiency lost with conventional multithreaded architectures. Instruction packet splitting allows an instruction bundle to be partially issued in one cycle, with the remainder of the bundle issued during a subsequent cycle. Thus, the present invention provides greater utilization of hardware resources (such as the functional units) and a lower elapsed time across a workload comprising multiple threads.

[0013] There are times when instruction packets cannot be split without violating the semantics of the instruction packet assembled by the compiler. In particular, the input value of a register is assumed to be the same for instructions in a packet, even if the register is modified by one of the instructions in the packet. If the packet is split, and a source register for one of the instructions in the second part of the packet is modified by one of the instructions in the first part of the packet, then the compiler semantics will be violated.

[0014] Thus, the present invention utilizes a packet split identification bit that allows hardware to efficiently determine when it is permissible to split an instruction packet. Instruction packet splitting increases throughput across all instruction threads, and reduces the number of cycles that the functional units rest idle. The allocation hardware of the present invention assigns as many instructions from each packet as will fit on the available functional units, rather than allocating all instructions in an instruction packet at one time, provided the packet split identification bit has not been set. Those instructions that cannot be allocated to a functional units are retained in a ready-to-run register. On subsequent cycles, instruction packets in which all instructions have been issued to functional units are updated from their thread's instruction stream, while instruction packets with instructions that have been held are retained. The functional unit allocation logic can then assign instructions from the newly-loaded instruction packets as well as instructions that were not issued from the retained instruction packets.

[0015] The present invention utilizes a compiler to detect parallelism in a multithreaded processor architecture. Thus, a multithreaded VLIW architecture is disclosed that exploits program parallelism by issuing multiple instructions, in a similar manner to single threaded VLIW processors, from a single program sequencer, and also supporting multiple program sequencers, as in simultaneous multithreading but with reduced complexity in the issue logic, since a dynamic determination is not required.

[0016] A more complete understanding of the present invention, as well as further features and advantages of the present invention, will be obtained by reference to the following detailed description and drawings.

Brief Description of the Drawings

[0017]

FIG. 1 illustrates a conventional generalized microprocessor architecture;
FIG. 2 is a schematic block diagram of a conventional superscalar processor architecture;
FIG. 3 is a program fragment illustrating the independence of operations;
FIG. 4 is a schematic block diagram of a conventional very long instruction word (VLIW) processor architecture;
FIG. 5 is a schematic block diagram of a conventional multithreaded processor;
FIG. 6 illustrates a multithreaded VLIW processor in accordance with the present invention;
FIG. 7A illustrates a conventional pipeline for a multithreaded processor;
FIG. 7B illustrates a pipeline for a multithreaded processor in accordance with the present invention;
FIG. 8 is a schematic block diagram of an implementation of the allocate stage of FIG. 7B;
FIG. 9 illustrates the execution of three threads TA-TC for a conventional multithreaded implementation, where threads B and C have higher priority than thread A;
FIGS. 10A and 10B illustrate the operation of instruction packet splitting in accordance with the present invention;
FIG. 11 is a program fragment that may not be split in accordance with the present invention;
FIG. 12 is a program fragment that may be split in accordance with the present invention;
FIG. 13 illustrates a packet corresponding to the program fragment of FIG. 12 where the instruction splitting bit has been set;
FIG. 14 is a program fragment that may not be split in accordance with the present invention; and
FIG. 15 illustrates a packet corresponding to the program fragment of FIG. 14 where the instruction splitting bit has not been set.

Detailed Description

[0018] FIG. 6 illustrates a Multithreaded VLIW processor 600 in accordance with the present invention. As shown in FIG. 6, there are three instruction threads, namely, thread A (TA), thread B (TB) and thread C (TC), each operating at instruction number *n*. In addition, the illustrative Multithreaded VLIW processor 600 includes nine functional units 620-1 through 620-9, which can be allocated independently to any thread TA-TC. Since the number of instructions across the illustrative three threads TA-TC is nine and the illustrative number of available functional units 620 is also nine, then each of the instructions from all three threads TA-TC can issue their instruction packets in one cycle and move onto instruction *n+1* on the subsequent cycle.

[0019] It is noted that there is generally a one-to-one correspondence between instructions and the operation specified thereby. Thus, such terms are used interchangeably herein. It is further noted that in the situation where an instruction specifies multiple operations, it is assumed that the multithreaded VLIW processor 600 includes one or more multiple-operation functional units 620 to execute the instruction specifying multiple operations. An example of an architecture where instructions specifying multiple operations may be processed is a complex instruction set computer (CISC).

[0020] The present invention allocates instructions to functional units to issue multiple VLIW instructions to multiple functional units in the same cycle. The allocation mechanism of the present invention occupies a pipeline stage just before arguments are dispatched to functional units. Thus, FIG. 7A illustrates a conventional pipeline 700 comprised of a fetch stage 710, where a packet is obtained from memory, a decode stage 720, where the required functional units and registers are identified for the fetched instructions, and an execute

stage 730, where the specified operations are performed and the results are processed.

[0021] Thus, in a conventional VLIW architecture, a packet containing up to K instructions is fetched each cycle (in the fetch stage 710). Up to K instructions are decoded in the decode stage 720 and sent to (up to) K functional units (FUs). The registers corresponding to the instructions are read, the functional units operate on them and the results are written back to registers in the execute stage 730. It is assumed that up to three registers can be read and up to one register can be written per functional unit.

[0022] FIG. 7B illustrates a pipeline 750 in accordance with the present invention, where an allocate stage 780, discussed further below in conjunction with FIG. 8, is added for the implementation of multithreaded VLIW processors. Generally, the allocate stage 780 determines how to group the operations together to maximize efficiency. Thus, the pipeline 750 includes a fetch stage 760, where up to N packets are obtained from memory, a decode stage 770, where the functional units and registers are identified for the fetched instructions (up to N*K instructions), an allocate stage 780, where the appropriate instructions are selected and assigned to the FUs, and an execute stage 790, where the specified operations are performed and the results are processed.

[0023] In the multithreaded VLIW processor 600 of the present invention, up to N threads are supported in hardware. N thread contexts exist and contain all possible registers of a single thread and all status information required. A multithreaded VLIW processor 600 has M functional units, where M is greater than or equal to K. The modified pipeline stage 750, shown in FIG. 7B, works in the following manner. In each cycle, up to N packets (each containing up to K instructions) are fetched at the fetch stage 760. The decode stage 770 decodes up to N*K instructions and determines their requirements and the registers read and written. The allocate stage 780 selects M out of (up to) N*K instructions and forwards them to the M functional units. It is assumed that each functional unit can read up to 3 registers and write one register. In the execute stage 790, up to M functional units read up to 3*M registers and write up to M registers.

[0024] The allocate stage 780 selects for execution the appropriate M instructions from the (up to) N * K instructions that were fetched and decoded at stages 760 and 770. The criteria for selection are thread priority or resource availability or both. Under the thread priority criteria, different threads can have different priorities. The allocate stage 780 selects and forwards the packets (or instructions from packets) for execution belonging to the thread with the highest priority according to the priority policy implemented. A multitude of priority policies can be implemented. For example, a priority policy for a multithreaded VLIW processor supporting N contexts (N hardware threads) can have N priority levels. The highest priority thread in the processor is allocated before any other thread. Among threads with equal priority, the thread that waited the longest for allocation is preferred.

[0025] Under the resource availability criteria, a packet (having up to K instructions) can be allocated only if the resources (functional units) required by the packet are available for the next cycle. Functional units report their availability to the allocate stage 780.

[0026] FIG. 8 illustrates a schematic block diagram of an implementation of the allocate stage 780. As shown in FIG. 8, the hardware needed to implement the allocate stage 780 includes a priority encoder 810 and two crossbar switches 820, 830. Generally, the priority encoder 810 examines the state of the multiple operations in each thread, as well as the state of the available functional units. The priority encoder 810 selects the packets that are going to execute and sets up the first crossbar switch 820 so that the appropriate register contents are transferred to the functional units at the beginning of the next cycle. The output of the priority encoder 810 configures the first crossbar switch 820 to route data from selected threads to the appropriate functional units. This can be accomplished, for example, by sending the register identifiers (that include a thread identifier) to the functional units and letting the functional units read the register contents via a separate data network and using the crossbar switch 810 to move the appropriate register contents to latches that are read by the functional units at the beginning of the next cycle

[0027] Out of the N packets that are fetched by the fetch stage 760 (FIG. 7B), the priority encoder 810 selects up to N packets for execution according to priority and resource availability. In other words, the priority encoder selects the highest priority threads that do not request unavailable resources for execution. It then sets up the first crossbar switch 810. The input crossbar switch 810 routes up to 3K*N inputs to up to 3*M outputs. The first crossbar switch 810 has the ability to transfer the register identifiers (or the contents of the appropriate registers) of each packet to the appropriate functional units.

[0028] Since there are up to N threads that can be selected in the same cycle and each thread can issue a packet of up to K instructions and each instruction can read up to 3 registers there are 3K*N register identifiers to select from. Since there are only M functional units and each functional unit can accept a single instruction, there are only 3M register identifiers to be selected. Therefore, the crossbar switch implements a 3K*N to 3M routing of register identifiers (or register contents).

[0029] The output crossbar switch 830 routes M inputs to N*M or N*K outputs. The second crossbar switch 830 is set up at the appropriate time to transfer the results of the functional units back to the appropriate registers. The second crossbar switch 830 can be implemented as a separate network by sending the register identifiers (that contain a thread identifier) to the functional units. When a functional unit computes a result,

the functional unit routes the result to the given register identifier. There are M results that have to be routed to up to N threads. Each thread can accept up to K results. The second crossbar switch 830 routes M results to N*K possible destinations. The second crossbar switch 830 can be implemented as M buses that are connected to all N register files. In this case, the routing becomes M results to N*M possible destinations (if the register files have the ability to accept M results).

[0030] In a conventional single-threaded VLIW architecture, all operations in an instruction packet are issued simultaneously. There are always enough functional units available to issue a packet. When an operation takes multiple cycles, the instruction issue logic may stall, because there is no other source of operations available. In a multithreaded VLIW processor in accordance with the present invention, on the other hand, these restrictions do not apply.

[0031] FIG. 9 illustrates the execution of three threads TA-TC for a conventional multithreaded implementation (without the benefit of the present invention), where threads B and C have higher priority than thread A. Since thread A runs at the lowest priority, its operations will be the last to be assigned. As shown in FIG. 9, five functional units 920 are assigned to implement the five operations in the current cycle of the higher priority threads TB and TC. Thread A has four operations, but there are only two functional units 920 available. Thus, thread A stalls for a conventional multithreaded implementation.

[0032] In order to maximize throughput across all threads, and minimize the number of cycles that the functional units rest idle, the present invention utilizes instruction packet splitting. Instead of allocating all operations in an instruction packet at one time, the allocation hardware 780, discussed above in conjunction with FIG. 8, assigns as many operations from each packet as will fit on the available functional units. Those operations that will not fit are retained in a ready-to-run register 850 (FIG. 8). On subsequent cycles, instruction packets in which all operations have been issued to functional units are updated from their thread's instruction stream, while instruction packets with operations that have been held are retained. The functional unit allocation logic 780 can then assign operations from the newly-loaded instruction packets as well as operations that were not issued from the retained instruction packets.

[0033] The operation of instruction packet splitting in accordance with the present invention is illustrated in FIGS. 10A and 10B. In FIG. 10A, there are three threads, each with an instruction packet from location *n* ready to run at the start of cycle *x*. Thread A runs at the lowest priority, so its operations will be the last to be assigned. Threads B and C require five of the seven available functional units 1020 to execute. Only two functional units 1020-2 and 1020-6 remain, so only the first two operations from thread A are assigned to execute. All seven functional units 1020 are now fully allocated.

[0034] At the completion of cycle x, the instruction packets for threads B and C are retired. The instruction issue logic associated with the threads replaces the instruction packets with those for address *n+1*, as shown in FIG. 10B. Since the instruction packet for thread A is not completed, the packet from address *n* is retained, with the first two operations marked completed. On the next cycle, *x+1*, illustrated in FIG. 10B, the final two operations from thread A are allocated to functional units, as well as all the operations from threads B and C. Thus, the present invention provides greater utilization of hardware resources (i.e., the functional units 1020) and a lower elapsed time across a workload comprising the multiple threads.

[0035] An instruction packet cannot be split without verifying that splitting would violate the semantics of the instruction packet assembled by the compiler. In particular, the input value of a register is assumed to be the same for instructions in a packet, even if the register is modified by one of the instructions in the packet. If the packet is split, and a source register for one of the instructions in the second part of the packet is modified by one of the instructions in the first part of the packet, then the compiler semantics will be violated. This is illustrated in the program fragment 1110 in FIG. 11:

[0036] As shown in FIG. 11, if instructions in locations L1, L2 and L3 are assembled into an instruction packet, and R0 = 0, R1 = 2, and R2 = 3 before the packet executes, then the value of R0 will be 5 after the packet completes. On the other hand, if the packet is split and instruction L1 executes before L3, then the value of R0 will be 2 after the packet completes, violating the assumptions of the compiler.

[0037] One means to avoid packet splitting that would violate program semantics is to add hardware to the instruction issue logic that would identify when destination registers are used as sources in other instructions in an instruction packet. This hardware would inhibit packet splitting when one of these read-after-write hazards exists. The mechanism has the disadvantage of requiring additional hardware, which takes area resources and could possibly impact critical paths of a processor and therefore reduce the speed of the processor.

[0038] A compiler can easily detect read-after-write hazards in an instruction packet. It can therefore choose to never assemble instructions with these hazards into an instruction packet. The hardware would then be forced to run these instructions serially, and thereby avoid the hazard. Any instruction packet that had a read-after-write hazard would be considered in error, and the architecture would not guarantee the results. Although safe from semantic violations, this technique has the disadvantage that it does not exploit potential parallelism in a program, since the parallelism available in the instruction packet with a hazard is lost, even if the underlying hardware would not have split the packet.

[0039] The present invention combines compiler

knowledge with a small amount of hardware. In the illustrative implementation, a single bit, referred to as the split bit, is placed in the prefix of a multi-instruction packet to inform the hardware that this packet cannot be split. Since the compiler knows which packets have potential read-after-write hazards, it can set this bit in the packet prefix whenever a hazard occurs. At run-time, the hardware will not split a packet with the bit set, but instead will wait until all instructions in the packet can be run in parallel. This concept is illustrated in FIGS. 12 through 15.

[0040] The compiler detects that the three instruction sequence 1210 in FIG. 12 can be split safely, so the split bit is set to 1, as shown in FIG. 13. In FIG. 14, on the other hand, the three instruction sequence 1410 cannot be split, since there is a read-after-write hazard between instructions L1 and L3. The split bit is therefore set to 0, as shown in FIG. 15

[0041] It is to be understood that the embodiments and variations shown and described herein are merely illustrative of the principles of this invention and that various modifications may be implemented by those skilled in the art without departing from the scope and spirit of the invention.

Claims

1. A multithreaded very large instruction word (VLIW) processor, comprising:

   a plurality of functional units for executing a plurality of instructions from a multithreaded instruction stream, said instructions being grouped into packets by a compiler, said compiler including an indication in said packet of whether said instructions in said packet may be split; and
   an allocator that selects instructions from said instruction stream and forwards said instructions to said plurality of functional units, said allocator assigning instructions from at least one of said instruction packets to a plurality of said functional units if said indication indicates said packet may be split.

2. The multithreaded very large instruction word (VLIW) processor of claim 1, wherein said indication is a split bit.

3. The multithreaded very large instruction word (VLIW) processor of claim 1, wherein said allocator assigns as many instructions from a given instruction packet as permitted by an availability of said functional units.

4. The multithreaded very large instruction word (VLIW) processor of claim 1, further comprising a register for storing for execution in a later cycle an indication of those instructions from a given instruction packet that cannot be allocated to a functional unit in a given cycle.

5. The multithreaded very large instruction word (VLIW) processor of claim 4, wherein instruction packets in which all instructions have been issued to functional units are updated from the instruction stream of said thread.

6. The multithreaded very large instruction word (VLIW) processor of claim 4, wherein instruction packets with instructions indicated in said register are retained.

7. A method of processing instructions from a multithreaded instruction stream in a multithreaded very large instruction word (VLIW) processor, comprising the steps of:

   executing said instructions using a plurality of functional units, said instructions being grouped into packets by a compiler, said compiler including an indication in said packet of whether said instructions in said packet may be split; and
   assigning instructions from at least one of said instruction packets to a plurality of said functional units if said indication indicates said packet may be split; and
   forwarding said selected instructions to said plurality of functional units.

8. The method of claim 7, wherein said indication is a split bit.

9. The method of claim 7, wherein said assigning step assigns as many instructions from a given instruction packet as permitted by an availability of said functional units.

10. The method of claim 7, further comprising the step of storing for execution in a later cycle an indication of those instructions from a given instruction packet that cannot be allocated to a functional unit in a given cycle.

11. The method of claim 10, wherein instruction packets in which all instructions have been issued to functional units are updated from the instruction stream of said thread.

12. The method of claim 10, wherein instruction packets with instructions indicated in said register are retained.

13. An article of manufacture for processing instruc-

tions from an instruction stream having a plurality of threads in a multithreaded very large instruction word (VLIW) processor, comprising:

a computer readable medium having computer readable program code means embodied thereon, said computer readable program code means comprising program code means for causing a computer to:

execute said instructions using a plurality of functional units, said instructions being grouped into packets by a compiler, said compiler including an indication in said packet of whether said instructions in said packet may be split; and

assign instructions from at least one of said instruction packets to a plurality of said functional units if said indication indicates said packet may be split; and

forward said selected instructions to said plurality of functional units.

14. A compiler for a multithreaded very large instruction word (VLIW) processor, comprising:

a memory for storing computer-readable code; and

a processor operatively coupled to said memory, said processor configured to:

translate instructions from a program into a machine language;

group a plurality of said instructions into a packet; and

provide an indication with said packet indicating whether said instructions in said packet may be split.

15. The compiler of claim 14, wherein said instruction packet can be split provided the semantics of the instruction packet assembled by the compiler are not violated.

16. The compiler of claim 14, wherein said instruction packet can be split provided a source register for one of the instructions in a first part of said packet is not modified by one of the instructions in a second part of said packet.

17. An article of manufacture for compiling instructions from an instruction stream having a plurality of threads for use in a multithreaded very large instruction word (VLIW) processor, comprising:

a computer readable medium having computer readable program code means embodied thereon, said computer readable program code means comprising program code means for causing a computer to:

translate instructions from a program into a machine language;

group a plurality of said instructions into a packet; and

provide an indication with said packet indicating whether said instructions in said packet may be split.

FIG. 1
PRIOR ART

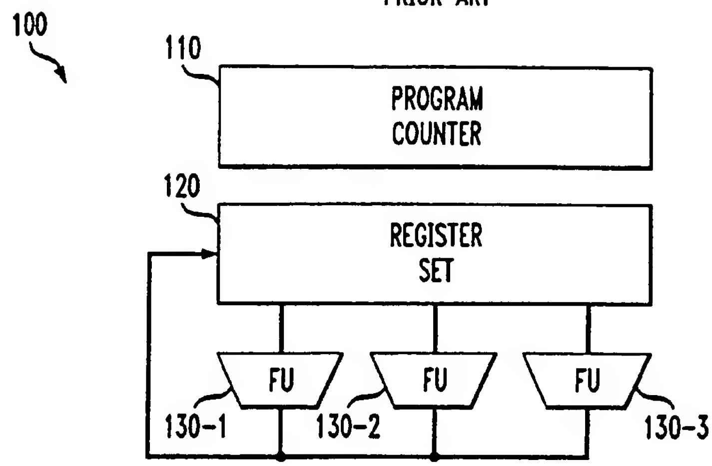


FIG. 2
PRIOR ART

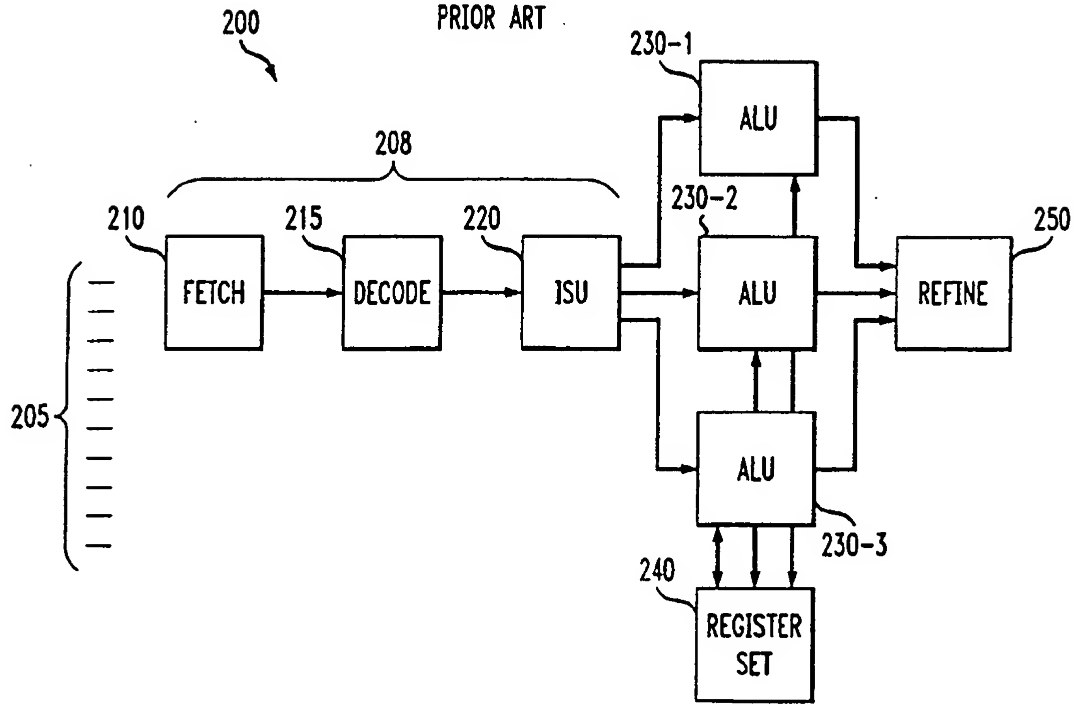


FIG. 3

310

```
L3: ADD R0, R1, R2
L2: SUB R3, R4, R2
L1: OR  R6, R1, R5
```

FIG. 4

PRIOR ART

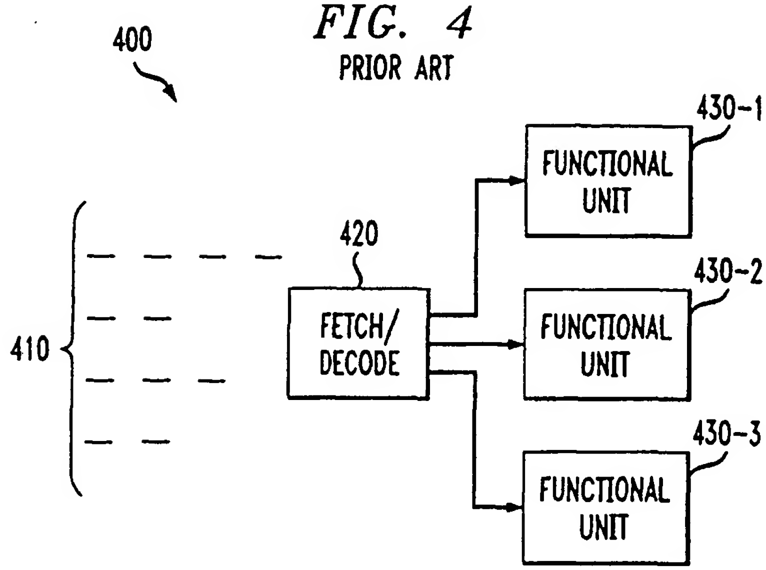


FIG. 5
PRIOR ART

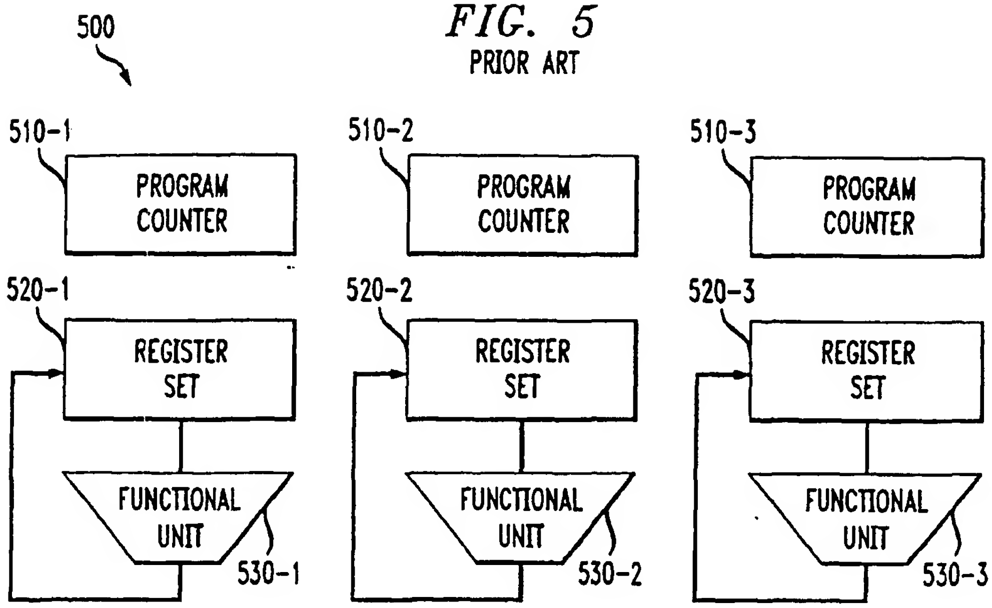


FIG. 6

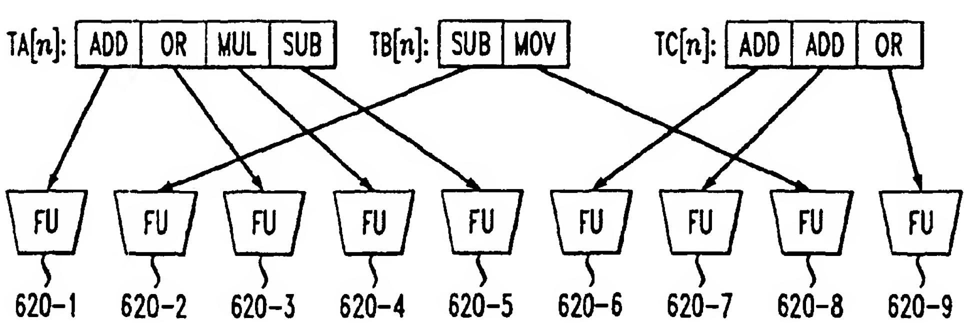


FIG. 7A

PRIOR ART

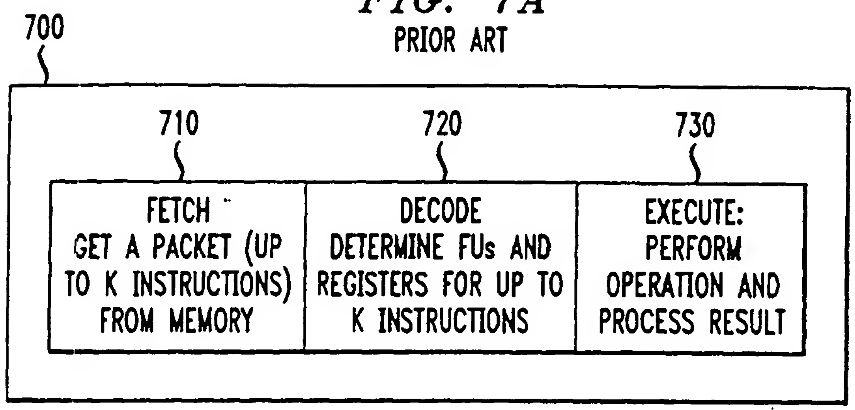


FIG. 7B

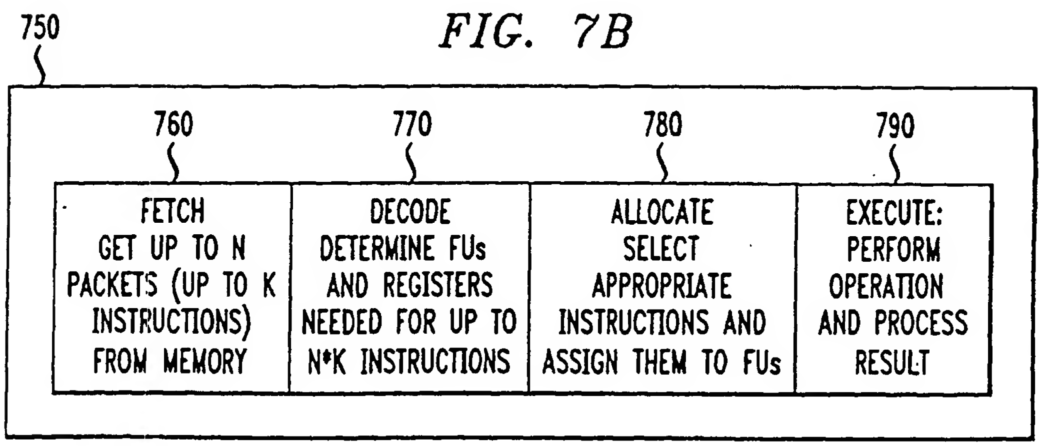


FIG. 8

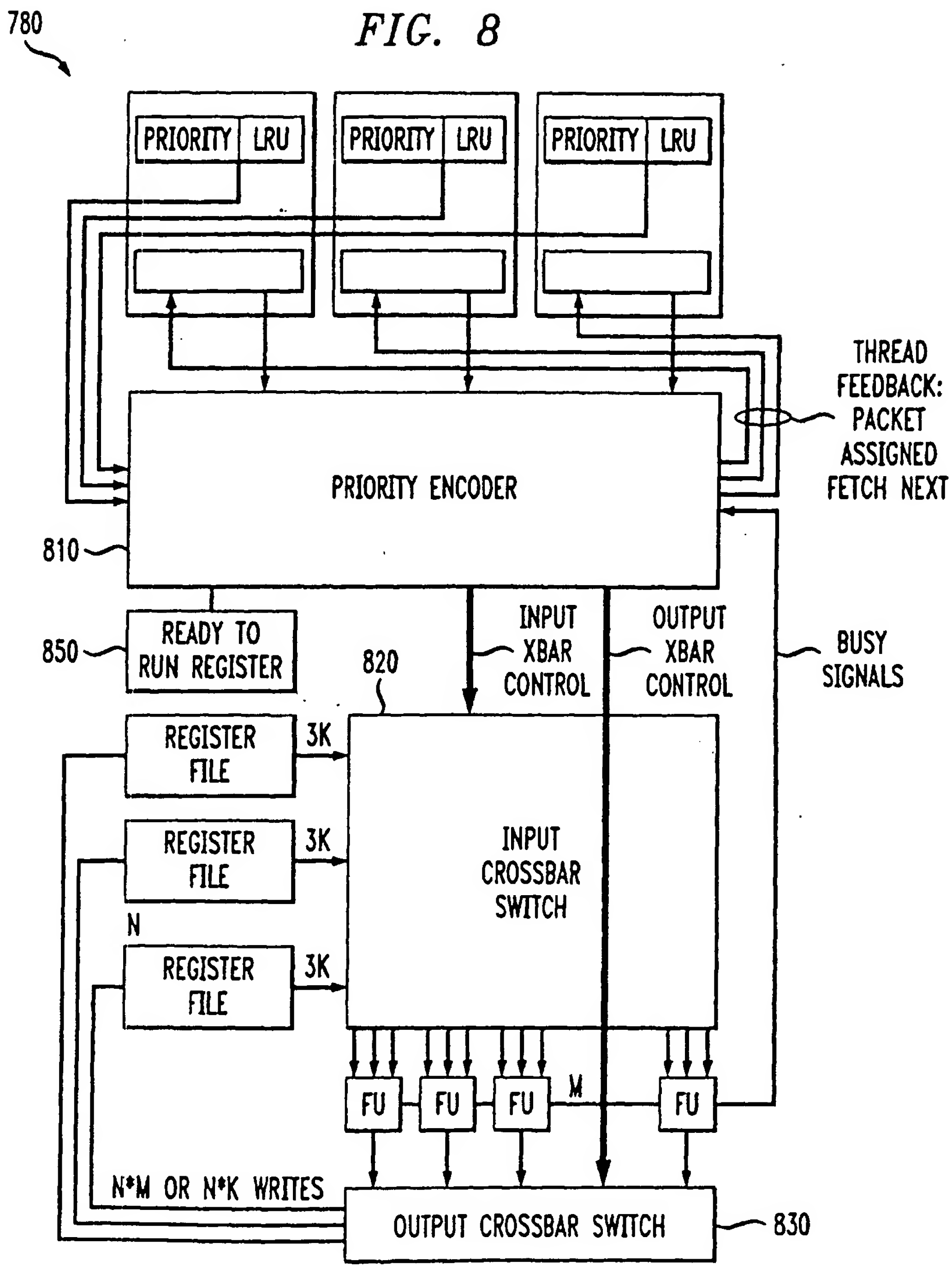

780
PRIORITY
LRU
PRIORITY
LRU
PRIORITY
LRU
THREAD FEEDBACK: PACKET ASSIGNED FETCH NEXT
PRIORITY ENCODER
810
850
READY TO RUN REGISTER
INPUT XBAR CONTROL
OUTPUT XBAR CONTROL
BUSY SIGNALS
820
REGISTER FILE
3K
REGISTER FILE
3K
N
REGISTER FILE
3K
INPUT CROSSBAR SWITCH
FU
FU
FU
M
FU
N*M OR N*K WRITES
OUTPUT CROSSBAR SWITCH
830

FIG. 9

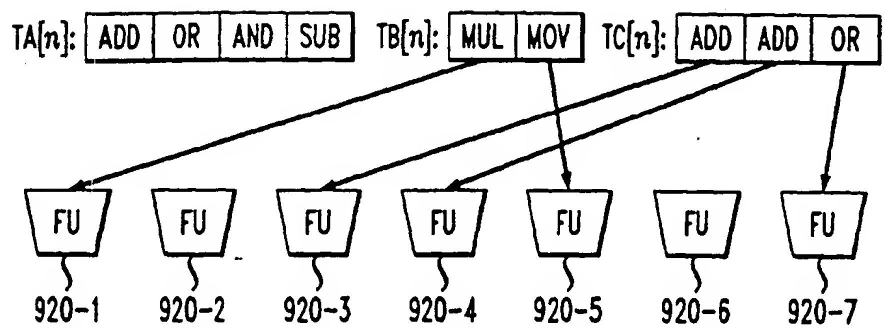


FIG. 10A

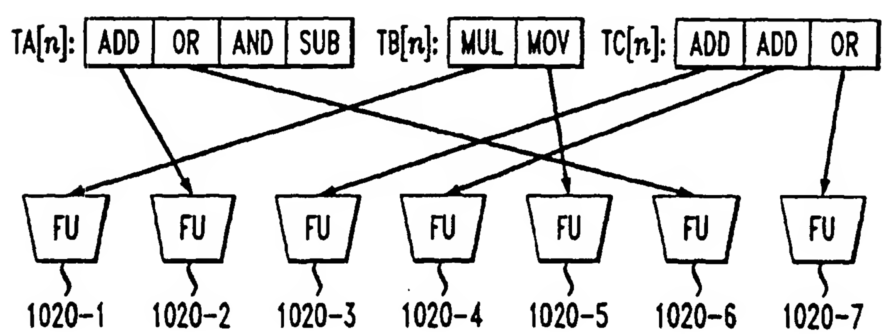


FIG. 10B

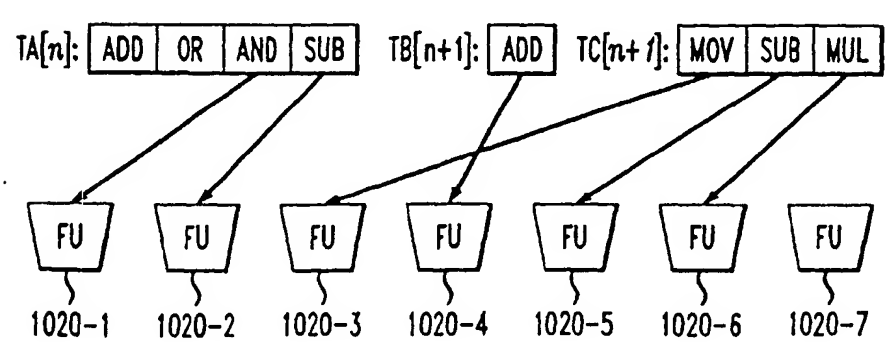


FIG. 11

```
1110 { L3: ADD R0,  R1, R2
       L2: SUB R3,  R4, R2
       L1: MOV R1,  R0
```

FIG. 12

```
1210 { L3: ADD R6,  R1, R2
       L2: SUB R3,  R4, R2
       L1: MOV R5,  R0
```

FIG. 13

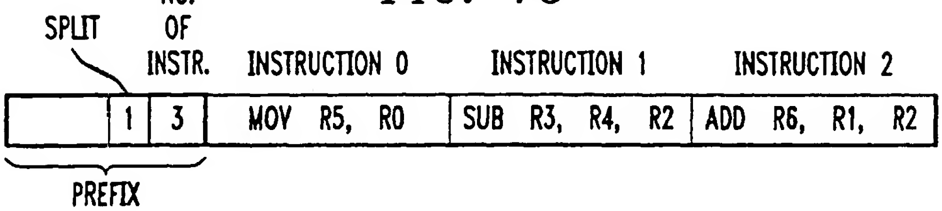
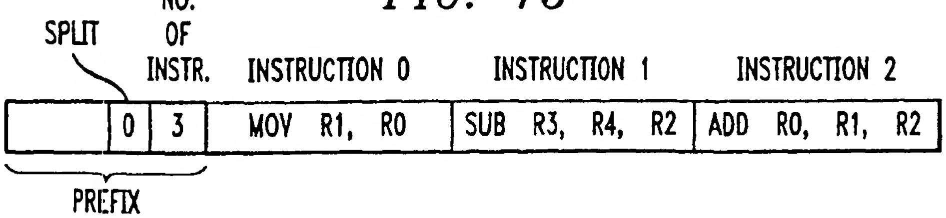


FIG. 14

```
1410 { L3: ADD R0,  R1, R2
       L2: SUB R3,  R4, R2
       L1: MOV R1,  R0
```

FIG. 15



European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 01 30 2951

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
A	US 5 574 939 A (KECKLER STEPHEN W ET AL) 12 November 1996 (1996-11-12) * abstract * * column 3, line 58 - column 4, line 8; figure 1 * * column 8, line 25 - line 57 * * column 9, line 29 - column 10, line 17 *	1-17	G06F9/38
A	US 5 404 469 A (CHUNG JIN-CHIN ET AL) 4 April 1995 (1995-04-04) * abstract * * column 6, line 35 - column 8, line 55 *	1-17	
P,A	EP 1 050 808 A (ST MICROELECTRONICS SA) 8 November 2000 (2000-11-08) * paragraph '0006! - paragraph '0009! * * paragraph '0032! - paragraph '0035! *	1,7,13, 14,17	
P,A	US 6 170 051 B1 (DOWLING ERIC M) 2 January 2001 (2001-01-02) * column 5, line 11 - line 38 * * column 10, line 56 - column 11, line 20 * * column 13, line 11 - line 30 *	1-17	

TECHNICAL FIELDS SEARCHED (Int.Cl.7)
G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	4 September 2001	Moraiti, M

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 01 30 2951

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

04-09-2001

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5574939 A	12-11-1996	WO 9427216 A	24-11-1994
US 5404469 A	04-04-1995	DE 4217012 A	26-08-1993
		JP 2928695 B	03-08-1999
		JP 7191847 A	28-07-1995
EP 1050808 A	08-11-2000	JP 2000330790 A	30-11-2000
US 6170051 B	02-01-2001	US 6163836 A	19-12-2000

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82

EXPRESS MAIL NO. EV347012631US

Improving Binary Compatibility in VLIW Machines through Compiler Assisted Dynamic Rescheduling

Morteza Biglari-Abhari, Kamran Eshraghian and Michael J. Liebelt+
Centre for Very High Speed Microelectronic Systems
Edith Cowan University, Joondalup, Western Australia, 6027
+Department of Electrical and Electronic Engineering,
University of Adelaide, Adelaide 5005, South Australia
email: m.biglari_abhari@cowan.edu.au

Abstract

One of the main problems that prevents extensive use of VLIW architectures for non-numeric programs is lack of object code (or binary) compatibility among different implementations of the same architecture. This is due to exposing all architectural features to generate code at compile time. New features of a VLIW machine may lead to incorrect results by executing the code compiled for the older machine. In this paper, a new approach to overcome this problem is presented, which we call dynamic VLIW generation (DVG). It is performed with the help of code annotation provided by the compiler, to reduce the complexity of the required hardware. In the DVG technique, operations are rescheduled for the new machine at the time of instruction cache miss repair. In this way, the rescheduler hardware is not located in the execution pipeline engine avoiding potentially longer cycle times. To simplify the dependency checking hardware, dependency information is encoded for each operation at compile time. This information can be combined into the final binary code, or may be provided as a separate file, which can be loaded into memory at execution time by the OS loader. In this technique operations can be rescheduled speculatively and a mechanism is presented to prevent destroying the contents of live registers. Experimental results show that the performance of rescheduled code using the DVG technique is about 10% worse than code compiled directly for the target processor.

1. Introduction

A Very Long Instruction Word (VLIW) processor [9] exploits ILP that has been extracted through compiler transformations. The advantage of these processors is their ability to exploit large amounts of ILP with relatively simple and fast control hardware. This is in comparison with the popular superscalar processors [12], in which ILP extraction is performed dynamically at run-time.

In VLIW processors, architecturally visible parallelism capability is exposed to the compiler. This includes an accurate semantic model of the processor, which defines the operation latencies, and a description of the allowed sets of independent operations in a VLIW instruction. The two main features of VLIW architectures are multiple-operation instruction (MultiOp) [18] and non-unit assumed operation latency (NUAL) [17].

In conventional RISC and superscalar processors, which are usually based on unit-assumed latency (UAL) execution semantics, it is assumed that previous operations in the sequence are committed at the time of source operand access for the current operation. If this is not the case (such as an out-of-order superscalar processor), or when the actual latency is greater than specified, additional hardware mechanisms such as register interlocks are provided to preserve the program correctness. Figure 1 shows a piece of code to illustrate differences in interpretation of program semantics between NUAL and UAL. In Figure 1 (a), operations 3 and 4 use the results of operations 1 and 2 respectively. In Figure 1 (b), due to UAL semantics, both operations 3 and 4 use the result of operation 2 and the destination register of operation 1 is overwritten by operation 2. In this manner, the semantics of the same code are different in Figures 1 (a) and (b).

NUAL provides more efficiency in code optimization and generation for the compiler. As the details of the target processor are exposed to the compiler, the resource usage of the target processor can be more efficiently scheduled by the compiler. However, this is at the expense of introducing some problems when the control flow at run-time is not the same as assumed at compile time. Unexpected events like exceptions dynamically change the assumed program

1089-6503/00 $10.00 © 2000 IEEE

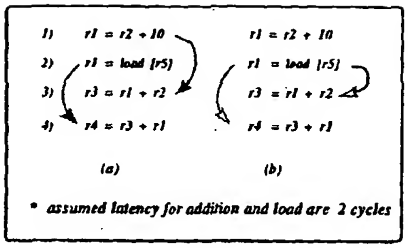


Figure 1: Comparison of the execution semantics for (a) NUAL and (b) UAL operations.

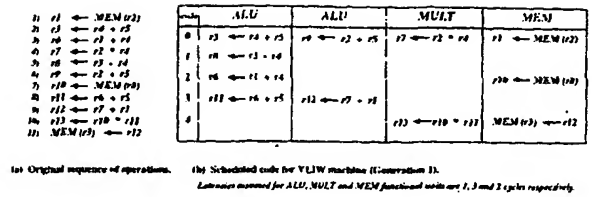


Figure 2: Scheduled code example for an issue-4 hypothetical VLIW machine (generation-1).

behavior due to control transfer to the exception handling routine, which causes the assumed latency of the scheduled operations to be non-deterministic. Hence, exact NUAL semantics are violated.

To handle this problem, two scheduling models of NUAL referred to as EQ (equals) model and LEQ (less than or equals) model have been defined [17]. An EQ NUAL operation accesses its operands at the specified time and writes back the result exactly at its latency time. In the LEQ model, if the operation latency is L, it is assumed the result is available between one and L cycles after launching the operation. Therefore, unexpected run-time events are handled more easily with LEQ NUAL semantics, while with EQ NUAL, hardware support is required to save the status of the processor before execution of exception handling code.

1.1 Object-Code Incompatibility

When the assumed architectural features such as operation latencies, the number of functional units and register file specifications are changed, object code compatibility is not preserved among different implementations of the same architecture. Figure 2 shows an example of a sequence of operations scheduled for a hypothetical issue-4 VLIW machine. In Figure 3, execution of the same code, when the latencies for MULT and MEM functional units have increased, leads to incorrect results. A decrease in latencies does not violate the program semantics for the LEQ model but, this is not the case for the EQ model. In the case of changes in the number of functional units, a narrower issue processor may execute a section of the VLIW instruction from the original VLIW code but, keeping correctness and respecting dependencies is not guaranteed. In a wider issue processor, the original code may be executed correctly but the additional new resources are not utilized.

In this paper, a new approach to overcome this problem is presented. It is performed with the help of code annotation provided by the compiler to reduce the complexity of the required hardware. The rest of this paper is organized as follows. In section 2, related works is reviewed. Our new approach is described in section 3. Differences between our approach and previous research is discussed in section 4. Section 5 presents the preliminary experimental results. Section 6 concludes the paper and describes possible future works.

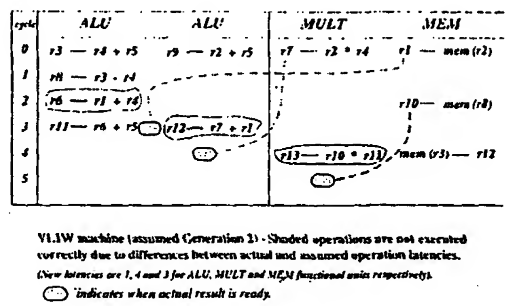


Figure 3: Incorrect interpretation of the code scheduled for VLIW (generation-1) in a new VLIW (generation-2) with different operation latencies.

2 Related Works

Several software and hardware approaches to overcome the binary incompatibility problem have been proposed by researchers. An application program which was compiled for one architecture can be converted to a new code for another architecture through binary translation. This is an off-line approach which does not affect the run-time behavior of the processor. This technique has been used for large scale migration of programs from one generation of machine to the next [19, 20].

Run-time rescheduling can be performed by a software method, or a hardware technique, or a combination of both. Figure 4 shows a run-time rescheduling technique through the operating system of the underlying machine. Two major projects have been reported which use this approach.

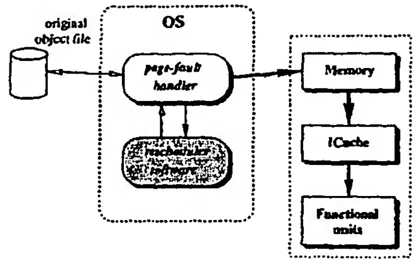


Figure 4: OS-based dynamic rescheduling.

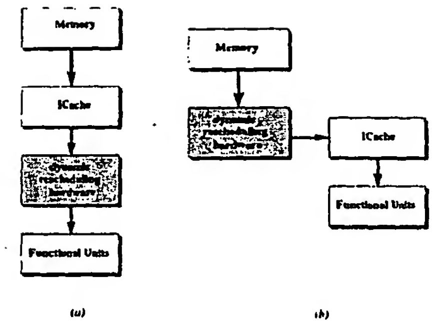


Figure 5: Outline of hardware-based techniques to achieve object code compatibility for different generations of the VLIW architecture. (a) Dynamic rescheduling at the execution pipeline path. (b) Dynamic rescheduling before moving operations to the instruction cache.

Conte and Sathaye described a dynamic rescheduling approach to reschedule code on a page-fault [6]. At the first-time page fault, the OS detects the difference between the current machine's characteristics and those assumed to generate the executable code. This page of code is rescheduled by special scheduling software that is invoked by the OS to generate a correct executable code for the current machine. When the same pages are accessed again, they are retrieved from the text swap space of the OS, where they were saved when displaced by new pages. A technique called the persistent rescheduled-page cache (PRC) was proposed to reduce the overhead when the same pages are accessed several times [7].

Another strategy based on the approach shown in Figure 4 was presented by Ebcioglu and Altman [8]. The aim of this work is to emulate an old architecture on a new architecture. It follows similar steps to those mentioned above, to generate a new schedule for a tree-based VLIW processor.

Figure 5 illustrates the general outline of hardware-based techniques to achieve object code compatibility among different generations of a VLIW architecture. In Figure 5 (a), dynamic rescheduling is performed in the execution pipeline path. Rau [17] presented dynamic scheduling for non-unit assumed operation latencies (NUAL) execution semantics in VLIW machines.

In Figure 5 (b), dynamic rescheduling is performed out of the execution pipeline path. One proposed approach in this category is based on a mechanism called the fill unit. The fill unit was originally proposed to form larger executable units from microoperations [15]. It was extended by Franklin and Smotherman [10] to form VLIW instructions dynamically from operations that can be issued together. This works as follows. The fill unit receives operations in the program order from the conventional instruction cache and decoder. A group of decoded operations which can be issued at the same time are formed in the fill unit buffer. When the buffer is full, it is written into an extra cache called the *shadow cache*. At each cycle, both the conventional instruction cache and the shadow cache are accessed and if the shadow cache is hit, the VLIW type instruction from the shadow cache is sent to the functional units. Otherwise, one operation fetched from the conventional I-Cache is executed.

The DIF (Dynamic Instruction Formating) machine [16] is another approach which can be considered to be based on Figure 5 (b). In the DIF machine, operations are executed for the first time by a separate processing engine referred to as the *primary engine* in [16]. At the same time, the dependency information is provided by the primary engine to translator hardware, which reschedules these executed operations as a DIF group. A DIF group is the unit of execution and consists of a sequence of VLIW instructions. DIF groups are stored in the DIF cache which is connected to another execution engine called the *parallel engine* and is similar to a VLIW execution pipeline. This approach is able to schedule operations speculatively above a few conditional branches.

Miss Path Scheduling (MPS) introduced by Banerjia and et al. [2] is also based on the approach of Figure 5 (b). In this technique, operations are rescheduled at cache miss time when the operations are received from a higher level of memory. The rescheduling hardware is able to schedule operations speculatively above conditional branches. Then, scheduled blocks are provided to an in-order execution engine. A scheduled block contains a set of VLIW-type instructions. The MPS technique uses one cache to hold operations fetched by the processor pipeline, in comparison with two different caches (shadow cache and instruction cache) required for the fill unit approach and the DIF method.

3 A New Approach

This section describes our proposed approach to overcoming the binary incompatibility problem between differ-

ent generations of VLIW machines. We call this *dynamic VLIW generation* (DVG). It requires ISA support and additional hardware, but this hardware is not located on the critical path of the execution pipeline. Operations are rescheduled at the cache miss repair. Thus, DVG is based on the approach in Figure 5 (b).

To simplify dependency checking hardware, some form of dependency information is encoded for each operation at compile time. Each operation has a *dependency_word*. This information can be combined into the final binary code or may be provided in a separate file, which can be loaded into memory by the OS loader at execution time.

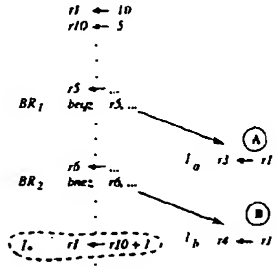
To schedule operations, the compiler generates a dependency graph to capture all dependency information in a scheduling region (such as a hyperblock). Transferring all of this information through the object file is not practical as it requires a large amount of disk space. Also, this information should be available in a suitable form to be used by a dynamic rescheduler with low overhead. Therefore, for DVG, the compiler provides a limited form of dependency information which will be processed at the time of instruction cache miss repair.

The simplest way of encoding dependency information is by using a bit vector for each group of operations. Each bit can represent a dependency to a previous operation based on its position in the bit vector. For example, if an operation has a dependency on the 14th operation before it, then bit 13 in the bit vector is set. This scheme however, increases the code size. If each operation is 32 or 64 bits, then having a 64-bit dependency_word is unacceptable. To reduce the size of the dependency_word, each bit may represent dependency to a packet of previous operations. For example, if each bit represents dependency to a packet of eight operations, keeping dependency information for 64 previous operations requires only 8 bits. This is at the expense of imposing more limitations at the time of rescheduling. As an example, if operation x is only dependent on one of the operations in the corresponding packet, it cannot be scheduled before all eight operations in that packet unless it is known which operation in that packet should precede it. This can be done by using special hardware to resolve the issue. Details are described in the following sections.

3.1 Speculative Scheduling

It has been shown that speculative scheduling is an essential technique to achieve higher amounts of ILP in general-purpose applications [13]. In order to perform speculative code motion in DVG, several issues should be considered.

The first issue is a method to predict conditional branches. In our scheme, two possibilities are available. One is an ISA extension, so that each conditional branch operation has an additional bit to indicate the prediction of its direction [11]. This bit is set if the branch is predicted taken. The prediction can be based on profile information or the heuristics used by the compiler. The other possible method is modifying the code layout, so that the most likely direction of a forward conditional branch is its fall-through path.



(a)

architectural index	L0	L1	L2	L3	active_L	SA
r1	-	10	-	-	01	00

(b)

architectural index	L0	L1	L2	L3	active_L	SA
r1	-	10	6	-	10	01

(c)

architectural index	L0	L1	L2	L3	active_L	SA
r1	6	10	-	-	00	10

(d)

- means don't care.
L0 to L3 indicate physical locations.
active_L shows the index of current physical location for the register.
SA is speculation adjustment.

Figure 6: An example of using the rotational remapping scheme in DVG with special status fields to restore registers' original state when a branch is mispredicted. (a) A sequence of operations to be rescheduled. (b) State of r1 before rescheduling of I_*. (c) State of r1 after moving I_* above BR_2. (d) State of r1 after moving I_* above BR_1.

The second issue is providing a mechanism to keep and restore the processor state when the prediction direction is not valid. Speculative motion of operations which write into a register is not allowed when the destination register is live on the other path. For this purpose, it is necessary to rename the architectural registers.

Register renaming involves mapping an architectural register into a physical register where the result of the operation is written. Following operations refer to this physical register to retrieve the value for their input operands. In a

typical hardware register renaming scheme in superscalar processors, a mapping table is looked up to determine the current mapping of the register when the contents of the register is read. The mapping table is updated when the register is written. A large number of ports is required to perform this process when multiple operations are executed concurrently.

To avoid this complexity, we extend a form of the *rotational remap* renaming scheme which was presented in [16]. In our scheme, the register file has several physical locations and status fields for each architectural register. The status fields indicate which physical location holds the most recent value and how the original physical location can be identified when a branch is mispredicted. Figure 6 shows an example to indicate how this scheme works.

A sequence of operations with two conditional branches is shown in Figure 6 (a). The aim is to move operation I_* above branches BR_1 and BR_2. Figure 6 (b) shows the state of register r1 if I_* is not scheduled above BR_1. After moving it up above BR_2, the state is changed to that shown in Figure 6 (c). The *active_L* field indicates the current physical location mapped to r1. *Speculation adjustment (SA)* determines how many locations should be moved to the left (considering a rotating remapping scheme) to reach the correct value of the architectural register when the branch is mispredicted. In the case of moving I_* above BR_2, the value in *speculation adjustment* field is set to 01, which means one shift to the left to access the original value of r1 if BR_2 is taken.

Figure 6 (d) shows the case when I_* is scheduled above BR_1. In this case, I_* is speculated over two conditional branches and if one of them is taken the original state is preserved through moving two locations to the left (based on *speculation adjustment* of 10). It should be noted that, no operation exists in the fall-through path of BR_1 to change r1. Otherwise, it would not be possible to schedule I_* above BR_1 due to that dependency constraint.

3.2 DVG Scheduling Algorithm

We propose the DVG approach in the context of predicated code based on hyperblock scheduling [14]. The beginning and end points of each hyperblock are encoded in the ISA by the compiler. Any code motion is performed in the hyperblock. This means that the original hyperblock is rescheduled for the new machine.

Figure 7 illustrates the DVG algorithm. At instruction cache miss time, the value of the PC is used to load at most N operations from the upper level of the memory hierarchy. N is implementation dependent. The issue-width of the processor and the implementation details of the DVG hardware are used to determine N, so that the amount of rescheduling overhead is tolerable.

```
// Input:  N (maximum numbers of operations to be rescheduled)
           PC (address of missed operation in the ICache)
// Output:  rescheduled code in the scheduling_buffer

last_scheduled_cycle = 0
for N operations
    use PC to fetch operation <op> and its dependency_word from memory
    if <op> is a branch then
          start_cycle = last_scheduled_cycle
    else
          start_cycle = 0
    use start_cycle and dependency_word to find the sch_cycle for <op>
    if valid sch_cycle not found then
          // this means that no resource available.
          terminate the scheduling process
    if scheduled <op> is speculative and has dst_reg then
          set SP_status of the register dst_reg
    place scheduled <op> in the scheduling_buffer
    if sch_cycle > last_scheduled_cycle then
          last_scheduled_cycle = sch_cycle
    if <op> is the last operation in the hyperblock then
          terminate the scheduling process
    increment PC (PC = PC + 4 in our case)
```

Figure 7: Main steps of the DVG scheduling algorithm.

After receiving each operation from the upper level of memory, it is scheduled and placed in the *scheduling buffer*. The scheduling buffer is similar to the reservation table in [2]. Its structure is like a matrix with *m* rows and *n* columns, where *m* is the maximum number of cycles (or the maximum number of generated VLIWs), and *n* is the issue-width of the processor. Also, a structure called the *operation scoreboard* is used to record the status of each processed operation. The cycle in which the result of each scheduled operation will be ready is recorded in the *operation scoreboard*.

Each architectural register has a counter which is loaded with the latency value of the operation writing into it. The counter is decremented each cycle when the execution of an operation is started until it is 0. Then, the *ready* bit is set. When a register is accessed for reading, if the result is not ready, all operations in the current VLIW are stalled until the required result is ready. The main purpose of the counter is to handle cases when the latency of an operation scheduled in the previous scheduling region is not fulfilled. At the same time, this mechanism preserves the processor state at the time of unexpected events which may increase the latency of an operation. The value of these counters may be saved and restored when necessary to keep the processor state.

When the maximum number of generated VLIWs is reached, or the last operation of a hyperblock is scheduled, or N operations are processed, the rescheduling process is terminated and the contents of the *scheduling buffer* are transferred to the instruction cache. The amount of time

required for this step depends on the structure of the instruction cache. In [2], two cache organizations for holding VLIWs were considered. These are the uncompressed I-Cache and the compressed banked I-Cache [5]. As the structure of the uncompressed I-Cache is similar to the *scheduling buffer*, transferring VLIWs from the *scheduling buffer* to the I-Cache does not require a long time in comparison to the compressed banked cache. In the latter, NOP operations are not included in the cache and special encoding is used to handle this. This results in longer time to transfer operations from the *scheduling buffer* to the I-Cache.

To schedule operations speculatively, the rotational remapping scheme is employed as discussed above. To prevent scheduling of a branch before operations which originally preceded it, the latest scheduled cycle is kept in a special register. Branches cannot be reordered. Also, an operation cannot be moved above a call subroutine and an indirect jump. The ISA of our experimental VLIW architecture includes non-trapping version of excepting operations, so it does not require additional hardware to keep precise exception handling. Otherwise, similar hardware schemes to those used in any multiple-issue processor to achieve precise exception handling would be necessary.

cycle	slot 0		DW	slot 1		DW
0	I_1	r2 ← MEM(r1)	(0,0,0,0,0,0)	I_2	r4 ← MEM(r3)	(0,0,0,0,0,0)
1	I_3	r6 ← MEM(r5+8)	(0,0,0,0,0,0)			
2	I_4	r2 ← r2 + 1	(0,0,0,1,0,0)	I_5	r7 ← r4 - 1	(0,0,0,1,0,0)
3	I_6	r8 ← r6 - r2	(0,0,0,1,1,0)	I_7	r9 ← r2 * 3	(0,0,0,1,0,0)
4	I_8	r10 ← MEM(r11 + 4)	(0,0,0,0,0,0)	I_9	brz: r8 , L1	(0,0,0,1,0,0)
5	I_{10}	r6 ← r2 + r7	(1,1,0,0,0,0)	I_{11}	r12 ← MEM(r8)	(0,1,0,0,0,0)
6	I_{12}	r13 ← r10 + 2	(0,0,1,0,0,0)			
7	I_{14}	r14 ← r10 * r9	(1,1,0,0,0,0)	I_{15}	bnz: r13 , L2	(0,0,0,0,1,0)

A piece of code scheduled for an issue-2 VLIW processor.
L1 and L2 refer to the code outside of the region.
DW - Dependency_Word
Operation latency: load = 2, multiplication = 3, other = 1.

(a)

cycle	slot 0	slot 1	slot 2
0	I_1	I_2	I_3
1	I_4		
2	I_6	I_5	
3	I_9	I_7	I_{10}
4	I_{11}	I_{12}	I_8
5	I_{15}		I_{14}

Code scheduled for an issue-3 VLIW processor.
Operation latency: load = 2, multiplication = 2, other = 1.
Speculative operations are shown shaded.

(b)

Figure 8: An example of DVG rescheduling process. (a) Code scheduled for the old machine. (b) Code rescheduled for the new machine.

We use the example in Figure 8 to describe how the DVG approach works. Figure 8 (a) shows a code segment scheduled for an issue-2 VLIW processor which is considered as the old-machine. Operations I_1 to I_{15} are located consecutively in memory with their dependency words. In this example, predicate operands are assumed as "TRUE" and are not shown. Also, we assume each bit in the *dependency_word* is used to indicate a dependency to one previous operation.

Figure 8 (b) illustrates the scheduled operations for the new VLIW machine, which is an issue-3 processor and the latency for multiplication is reduced to 2 cycles in comparison to the old machine. The scheduling process is as follows.

When the cache miss occurs, the address of the operation resulting in the cache miss is used to get the operation from the higher level of memory hierarchy with its *dependency_word*. The *dependency_word* is applied to the *operation scoreboard* to find the earliest cycle for scheduling. Operations I_1, I_2 and I_3 do not have dependency on previous operations (as the region begins with I_1). Therefore, they can be scheduled in cycle 0 in the new machine. Then, operation I_4 is processed. It is dependent on I_1 so, referring to the *operation scoreboard* determines that I_4 can be scheduled in cycle 2. Other operations are processed and scheduled in this way. Operations I_{10}, I_{11} and I_{12} are speculative and the status fields in the related architectural registers are set appropriately as described before. When an excepting operation is speculated, its non-trapping version is used as the speculative operation.

In the case of backward branches, which are marked by the compiler, a limited form of loop unrolling can be applied which is dependent on N (the maximum number of operations to be rescheduled) and the distance between the branch and its target.

4 Comparison with Related Works

The DVG approach, similarly to the fill unit [10], the DIF machine [16], and MPS [2], reschedules operations before placing them in the cache which is accessed by the parallel execution engine in the machine. The fill unit method lacks speculative scheduling and requires two instruction caches (the conventional I-Cache and the shadow cache). The DIF machine also employs two caches and has two different execution engines. The DVG method on the other hand requires one instruction cache. It is an extension of the conventional VLIW processor which reschedules operations dynamically at cache miss time. The most similar published work to the DVG method is the MPS method. In the MPS, operations are also rescheduled at the cache miss repair. The major difference between the DVG approach with the previous works (particularly with the MPS method) is the use of code annotation generated by the compiler to direct the rescheduling process. Also, a limited form of loop

unrolling can be performed for backward branches in the DVG.

5 Experimental Results

To evaluate the effectiveness of the DVG approach, experimental results were generated based on the following methodology.

We use our VLIW compiler [3], which generates predicated code through hyperblock scheduling [14] to compile benchmark programs. We modified the *mlcache* [21], which is a multi-lateral cache simulator so that at each cache miss at most N operations are fetched from the next level of the memory hierarchy and scheduled. The scheduled operations are placed into the appropriate locations of the instruction cache. A perfect data cache is assumed for all experiments. The instruction cache is a 64K direct mapped banked cache [5]. The assumed bandwidth between the I-Cache and the next level of memory is one word (32 bits) per cycle with six cycles latency. These are contemporary assumed values typical of current memory technology. To approximate the scheduling overhead, five additional cycles are added to the total miss repair latency. This assumption is reasonable, especially if pipelined hardware is employed.

To record the number of cycles, one cycle latency is assumed for each cache hit, and the processor stalls for the period of the cache miss repair. The number of execution cycles for the old machine and the speedup are obtained as follows.

The number of execution cycles for each benchmark program is estimated based on the static code schedules weighted by dynamic execution frequencies obtained from profiling. It was reported that the accuracy of results is better than 95% in comparison with simulation results assuming perfect caches [1,4]. Almost the same accuracy is expected in our case. This is because, a VLIW processor can be considered similar to the in-order issue processor used in [4] and [1].

Six SPEC95 integer programs are used as the benchmarks in this experiment. Each benchmark program is compiled with our compiler for the old machine, and the required code annotations (*dependency_word* and other operation marking such as backward branches and so on) are generated. Program traces (using the train inputs of the benchmarks) are generated in a proper format and are used by the *mlcache*. Speedup is calculated as the ratio of the number of cycles for basic block scheduling to the number of cycles for hyperblock scheduling. The results are generated with three different machine models as shown in Figure 9.

Figure 10 shows the speedup achieved through moving the older machine code to the new wider machine and applying the DVG technique. This indicates that the DVG technique uses additional resources in the new machine to

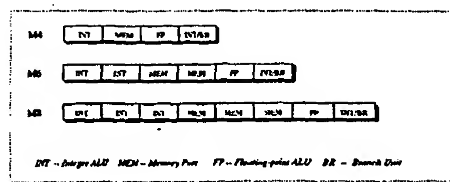


Figure 9: Issue slot configuration of three machine models.

improve the performance of the code compiled for the old machine.

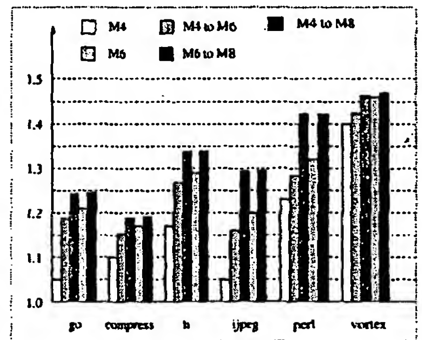


Figure 10: Speedup of the rescheduled old machine code on a wider new machine model.

Results in Figure 11 show the performance when the code complied for the old machine is rescheduled through the DVG method for the new machine in comparison to the code compiled for the new machine. For example, in Figure 11 (c) the M4 to M8 column indicates the reduction in speedup for the code originally compiled for M4 and rescheduled for M8 compared to the speedup of the code compiled for the M8 machine model.

The results show that performance of the DVG technique is higher for wider-issue processors. This is due to the more opportunities for scheduling provided when more resources are available.

6 Conclusions

In this paper, we have presented a novel approach to overcome the problem of binary incompatibility in VLIW machines. In this approach which is called DVG, operations are rescheduled for the new machine at the time of instruction cache miss repair. So, the rescheduler hardware is not located in the execution pipeline engine avoiding potentially longer cycle times. Experimental results show that the performance of rescheduled code using the DVG tech-

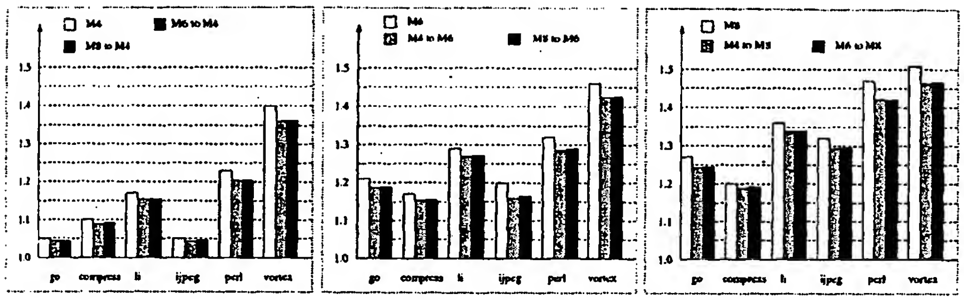


Figure 11: Speedup of the rescheduled code for machine models (a) M4, (b) M6 and (c) M8.

nique is about 10% worse than code compiled directly for the target processor.

The DVG technique was investigated in the context of predicated code. Future work may include research on the capability of the DVG for non-predicated code and evaluation of the rescheduling overhead and its impact on the performance.

References

[1] D. I. August, W. W. Hwu, and S. A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proc. of the 30th International Symposium on Microarchitecture*, December 1997.

[2] S. Banerjia, K. N. Menezes, S. W. Sathaye, and T. M. Conte. MPS: Miss-path Scheduling for Multiple Issue Processors. *IEEE Transactions on Computers*, 47(12), December 1998.

[3] M. Biglari-Abhari, M. J. Liebelt, and K. Eshraghian. Implementing a VLIW Compiler: Motivation and Trade-offs. In *Proc. of the Third Australasian Computer Architecture Conference - ACAC'98*, pages 37–46, Perth, Western Australia, February 2-3 1998. Published by Springer-Verlag.

[4] R. A. Bringmann. *Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[5] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 201–211, Paris, France, 1996.

[6] T. M. Conte and S. W. Sathaye. Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures. In *Proc. of the 28th International Symposium on Microarchitectures*, pages 208–218, December 1995.

[7] T. M. Conte, S. W. Sathaye, and S. Banerjia. A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibilty in VLIW Architectures. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 4–13, Paris, France, December 1996.

[8] K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Technical Report RC20538, IBM T. J. Watson Research Center, Yorktown Heights, NY, August 1996.

[9] J. A. Fisher. Trace Scheduling : A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[10] M. Franklin and M. Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. In *Proc. of the 27th International Symposium on Microarchitectures*, pages 162–171, San Jose, CA, December 1994.

[11] HP. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett Packard, Palo Alto, CA, 1994.

[12] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19-21 1992.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution using the Hyperblock. In *Proc. of the 25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[15] S. Melvin, M. Shebanow, and Y. Patt. Hardware Support for Large Atomic Units in Dynamically Scheduled Machines. In *Proc. of the 21th Annual Workshop on Microprogramming and Microarchitecture*, pages 60–66, San Diego, CA, December 1988.

[16] R. Nair and M. Hopkins. Expoliting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997.

[17] B. R. Rau. Dynamically Scheduled VLIW Processors. In *Proc. of the 26th International Symposium on Microarchitectures*, pages 80–90, 1993.

[18] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra-5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, 22:12–35, January 1989.

[19] G. M. Silberman and K. Ebcioglu. An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures. *Computer*, 26(6):39–56, June 1993.

[20] R. Sites and et. al. Binary Translation. *Communications of the ACM*, 36(2):69–81, 1993.

[21] E. S. Tam, J. A. Rivers, and E. S. Davidson. Flexible Timing Simulation of Multiple Cache Configurations. Technical Report CSE-TR-348-97, University of Michigan, November 1997.